Open in app ↗ Sign up Sign in

Medium Search Write

# Synchronization of native and WebView sessions with iOS

Paul Hackenberger · Follow

Published in Axel Springer Tech · 15 min read · Feb 16, 2024

23 3

First things first: The base of (almost) all digital business ideas are user identification and offering services via purchase or subscription. Those two are the base logic of most applications, and essential for business success.

National Media & Tech uses a web-centered approach for session management based on JSON Web Token, OAuth 2.0 and HTTP Cookies.

This article focusses on the challenges to bring a primarly web-based session management to iOS.

> Don’t be confused by the structure of this article: It describes first the basics, then our Odyssee, and finally leads to the working solution. So read on my dear, to avoid our mistakes…

## The Basics: JWT, OAuth and HTTP Cookies

### JSON Web Token

JSON Web Token (JWT) is a standard (RFC 7519) for securely transmitting information as a JSON object, which can be digitally signed for integrity verification and trust. The signing can be done using a secret via HMAC or with RSA or ECDSA public/private key pairs. While JWTs can also be encrypted for confidentiality, the focus here is on signed tokens, which ensure the integrity of their claims without necessarily hiding them.

JWTs are particularly useful in two main scenarios:

1. **Authorization:** JWTs are commonly used for authorization purposes. After a user logs in, each subsequent request includes a JWT, granting access to authorized routes, services, and resources. They are instrumental in implementing Single Sign On (SSO) systems due to their lightweight nature and the ease with which they can be used across different domains.
2. **Information Exchange:** JWTs facilitate secure information transfer between parties, ensuring the sender's **identity** and **entitlements** through digital signatures and confirming the data's integrity since the signature encompasses both the header and payload.

In summary, JWTs offer a compact, flexible method for authentication and secure information exchange, leveraging digital signatures to ensure data integrity and authenticity.

By the way: Google Firebase Authentication is using the same approach.

### OAuth 2.0

OAuth 2.0 is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service, such as Facebook, GitHub, or Google. It works by delegating user authentication to the service that hosts the user account and authorizing third-party applications to access the user account. **Access tokens** are short-living and used to authorize API requests on behalf of a user, while **refresh tokens** are long-living and used to obtain new **access tokens** when the current access token expires without requiring the user to log in again.

### HTTP Cookies

Different types of cookies can be used for different purposes.
A persistent cookie is designed to store data on a user's browser until a specified expiration date or time period has passed. Unlike session cookies, which are deleted when the browser closes, persistent cookies remain across multiple sessions. They transmit information to the server every time the user revisits the associated website or whenever a user views a resource from that website on a different site, such as through an advertisement. This functionality makes persistent cookies useful for tracking users over time, allowing websites to keep users logged in between visits, enhancing user convenience by eliminating the need to repeatedly enter login details.
Using HTTP Cookies, and sending them via HTTP Header, is the de facto standard for session management in the web via browser; the browser takes care of alle the housekeeping of the cookies.

## Session Synchronization

## The USP of Native Apps

Meanwhile in the web everything is browser-based, choosing a native implementation for your app has mainly three advantages:

1. **Performance:** Native apps are generally faster and more responsive than WebView apps. They are optimized for the platform's hardware, allowing for smoother animations, quicker load times, and a more seamless experience overall.
2. **User Experience (UX):** Native iOS apps provide a superior user experience, as they adhere to iOS Human Interface Guidelines. This ensures that apps feel intuitive and familiar to users, leveraging the full range of gestures and interactions that iOS users expect.
3. **Access to Device Features:** Native apps have full access to the device's hardware and iOS features, such as the camera, GPS, accelerometer, push notifications, and more. This access allows for a more interactive and engaging app that can utilize the full potential of the device.

Even though you can achieve similar functionality by using Progressive Web Apps (PWA), or other multi-platform tools (e.g. Flutter), to achieve the optimal user experience on iPhone, a native app remains the superior option.

## Native Session Handling

So let's start the session with a native call, and not forget to send and receive all cookies issued, that contain the session information.

```
// The default session configuration includes automatic cookie handling, so you
let session = URLSession.shared
```

```
// Request
if let url = URL(string: "https://example.com") {
    var request = URLRequest(url: url)
    request.httpMethod = "GET" // or "POST", etc.

    let task = session.dataTask(with: request) { data, response, error in
        // Handle response and errors here
        if let error = error {
            print("Error: \(error.localizedDescription)")
            return
        }

        // Process the data/response here
        // Cookies are automatically handled
    }

    task.resume()
}
```

**dataTask(with:) | Apple Developer Documentation**

Creates a task that retrieves the contents of the specified URL.

developer.apple.com

## Key Points

- **Automatic Cookie Management:** With the default `URLSession` configuration, iOS automatically handles cookies. This includes storing cookies received in responses and automatically sending appropriate cookies with requests to the same server.
- **No Manual Cookie Handling Needed:** In this automatic mode, you don't need to manually add cookies to requests or save cookies from responses. The system takes care of these operations for you via the HTTPCookieStorage, simplifying session management.

- **Session Configuration:** Using `URLSession.shared` or creating a session with `URLSessionConfiguration.default` both provide automatic cookie handling. If you create a custom session configuration, ensure you don't disable automatic cookie handling unless you intend to manage cookies manually.

### Passing cookies to WKWebView... and back?

Now the session information stored in the cookies inside the HTTPCookieStorage needs to be passed to the WKWebView.

By default, cookies stored in `HTTPCookieStorage` are not automatically shared with `WKCookieStore` in a `WKWebView`, and vice versa. This separation occurs because `WKWebView` operates in a different process than your app, due to the WebKit's out-of-process architecture. This design enhances the performance and security of web content rendering but also means that cookie sharing between the web view and the app's HTTP requests isn't straightforward.

To insert Cookies into WKWebView's Cookie Store iterate over the cookies and add them to the WKWebView's cookie store. Since setCookie is asynchronous, if you need to perform an action after all cookies are set (like loading a web request), you can use `await` to wait for all asynchronous operations to complete.

```
for cookie in cookies {
    await webView.configuration.websiteDataStore.httpCookieStore.setCookie(cooki
}

group.notify(queue: .main) {
    // All cookies have been set, now you can load your request
    if let url = URL(string: "https://example.com") {
        let request = URLRequest(url: url)
        webView.load(request)
```

```
        }
    }
```

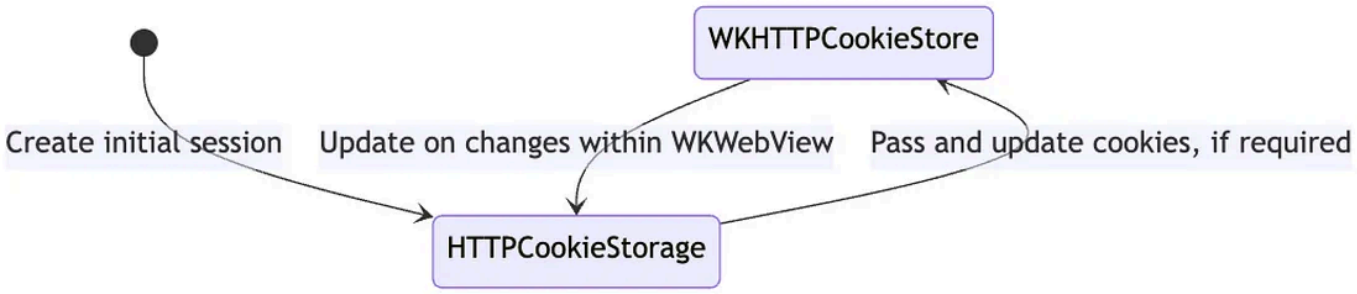
> *Ok — but what happens, if the cookies are updated by a native call in the background?*

In this case, we will have to pass the cookies again — and reload the WKWebView to make sure also the running web context re-reads all cookies.

> *Sounds doable, but what if the cookies were updated by WebView itself (e.g. JavaScript or set-cookie from server while navigating)?*

Mmh… good point! Then we have to subscribe to changes to WKHTTPCookieStore via WKHTTPCookieStoreObserver, and sync the changes back to the HTTPCookieStorage that is being used by native calls.

```
func copyCookiesToSharedStorage(from cookieStore: WKHTTPCookieStore) async {
    let cookies = await cookieStore.getAllCookiesAsync()
    for cookie in cookies {
        HTTPCookieStorage.shared.setCookie(cookie)
    }
}
```



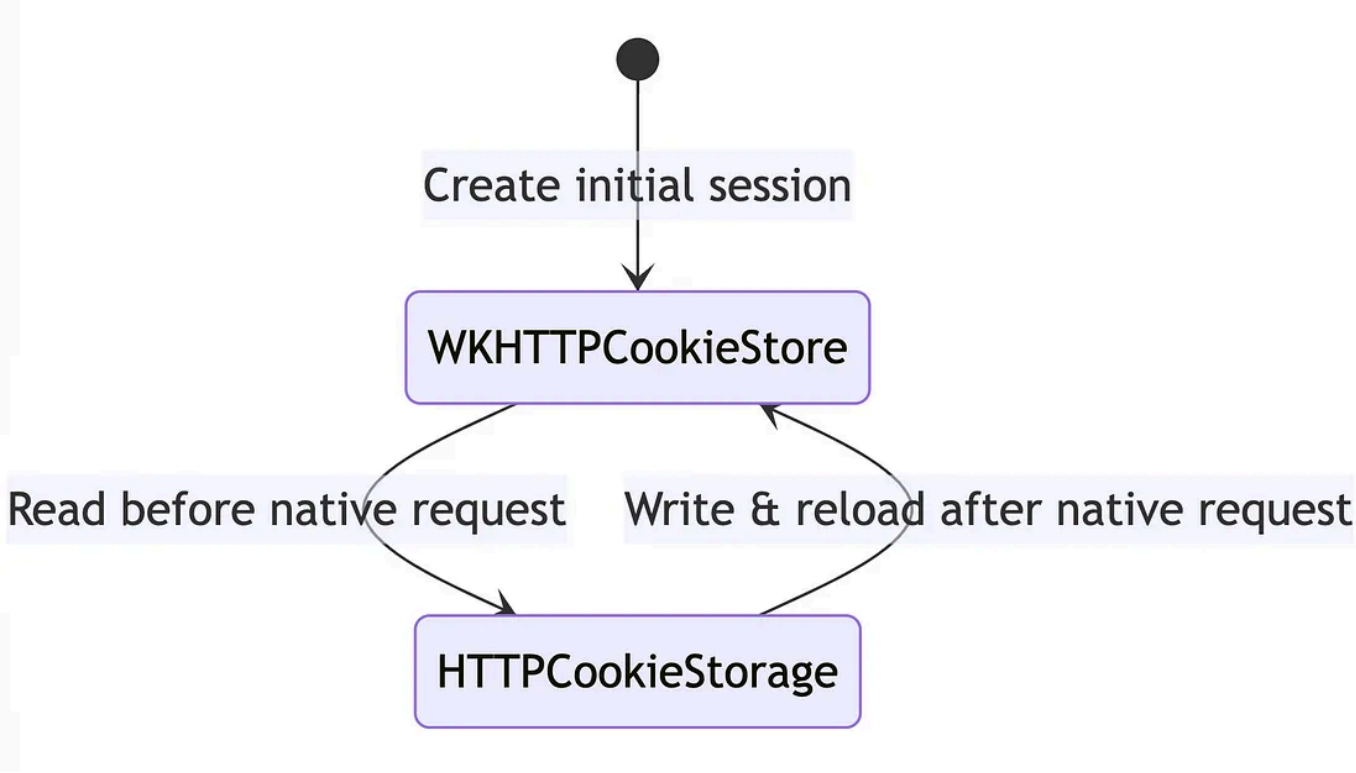

Full-sync HTTPCookieStorage and WKHTTPCookieStore (Mermaid)

## Single-Source of Truth for Cookie Storage

> One moment. This does not feel right.
> Maybe we put the cart before the horse!

There should be a single source of truth, and also with all the custom synching there could be race conditions occurring on simultaneous calls…

If we used the WKHTTPCookieStore as single point of truth, then it should be assured, that all changes that happen within a WebView are accessable to native calls and all native calls can directly write to WKHTTPCookieStore to make their cookie updates available to WebView in return.




WKHTTPCookieStore a single point of truth (Mermaid)

**Native Request always read and write to WKHTTPCookieStore**

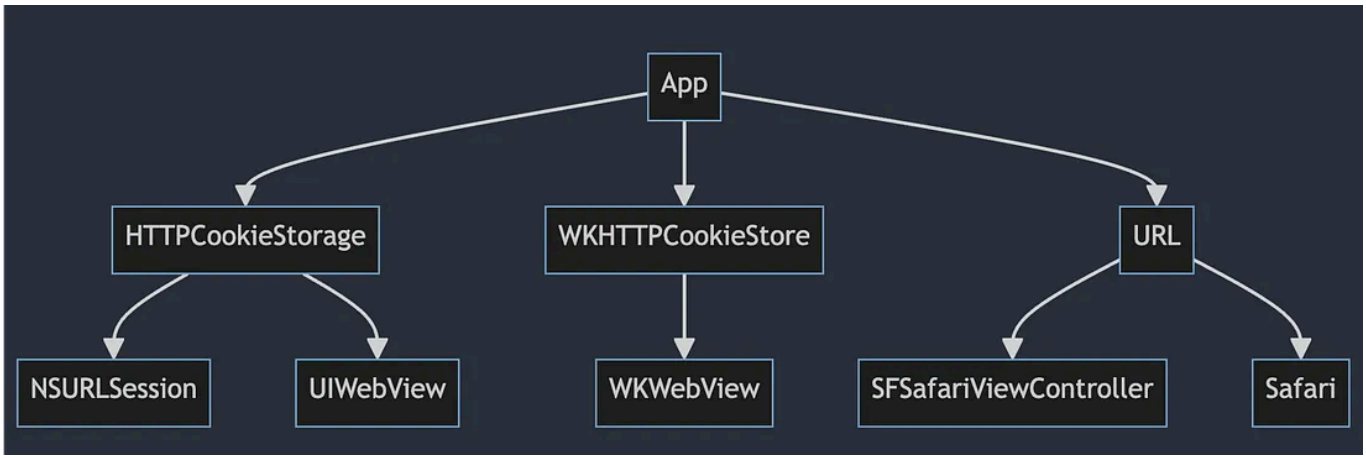
Now making WKHTTPCookieStore the single point of truth will provide all cookies automatically to all WKWebViews. The only thing we have to take care of is setting cookies explictly when doing native requests and write the resulting cookies back to the WKWebsiteDataStore and reload, to provide them to all WKWebViews.

```
// Retrieve cookies from the default data store asynchronously
WKWebsiteDataStore.default().httpCookieStore.getAllCookies { cookies in
    // Copy cookies to native cookie storage
    for cookie in cookies {
        HTTPCookieStorage.shared.setCookie(cookie)
    }

    // Execute request
    if let url = URL(string: "https://example.com") {
        var request = URLRequest(url: url)

        // URLSession per default uses cookies from HTTPCookieStorage.shared
```

```
            let task = session.dataTask(with: request) { data, response, error in
                // Handle errors
                if let error = error {
                    print("Request error: \(error)")
                    return
                }

                // Ensure we have a valid response
                guard let httpResponse = response as? HTTPURLResponse else {
                    print("Invalid response")
                    return
                }

                // Extract cookies from response headers and copy them to WKCookieSt
                if let headerFields = httpResponse.allHeaderFields as? [String: Stri
                    let cookies = HTTPCookie.cookies(withResponseHeaderFields: heade

                    // Copy cookies to WKCookieStorage
                    for cookie in cookies {
                        DispatchQueue.main.async {
                            WKWebsiteDataStore.default().httpCookieStore.setCookie(c
                                // Cookie is now set in WKHTTPCookieStore. You might
                            }
                        }
                    }
                }
            }
            task.resume()
        }
    }
```

## Side note

The described approach works fine for native requests and WKWebViews, but is unsuitable for SFSafariController or Safari, since the contexts of the Safari views are totally isolated from the app and therefore no cookies can be passed.




Mermaid

## Caveats of WKWebsiteDataStore

### Asynchronous load

The `WKWebsiteDataStore` is a part of the WebKit framework, providing a way to manage website data such as cookies, cache, and local storage for your web content. When you query `WKWebsiteDataStore` during `applicationDidLoad` or early in your app's lifecycle, it might appear empty due to the timing of data loading or initialization processes within the WebKit framework. Here are a few reasons why `WKWebsiteDataStore` might appear empty at `applicationDidLoad`:

1. Asynchronous Data Loading: The process of loading website data (like cookies and cache) into the `WKWebsiteDataStore` is **asynchronous. If you check the contents of the data store immediately upon application load, the data loading process may not have completed yet.**
2. Data Store Initialization: The `WKWebsiteDataStore` might not be fully initialized at the time of `applicationDidLoad`. This can depend on how your app initializes web content and accesses the data store.
3. No Data Yet: If your application is newly installed or has not yet created or received any website data (e.g., if a `WKWebView` has not yet loaded any content), the `WKWebsiteDataStore` will naturally appear empty.

4. Using the Default Data Store: If you're accessing the `default WKWebsiteDataStore`, it should contain the data for web content loaded through `WKWebView` instances in your app. However, timing and initialization might still play a role in what data appears to be available.

To overcome the asynchronous Data Loading problems mentioned, we ended up using `fetchDataRecords(ofTypes:completionHandler:)` rather than directly accessing the cookies, which then often were reported as empty.

**fetchDataRecords(ofTypes:completionHandler:) | Apple Developer Documentation**

Fetches the specified types of records from the data store.

developer.apple.com

```
let dataStore = WKWebsiteDataStore.default()
// Specify that you're interested in cookies
let dataTypes = Set([WKWebsiteDataTypeCookies])

dataStore.fetchDataRecords(ofTypes: dataTypes) { records in
    for record in records {
        print("\(record.displayName) has the following cookies:")

        for cookie in record.cookies {
            print(cookie)
        }
    }
}
```

### Isolated Cookies & session cookies

`WKWebView` — Each `WKWebView` instance can share the default persistence storage, have its own non-persistence storage (like incognito tab), or use a

specific persistence with ID to share data. See the `WKWebsiteDataStore` class for more information.

Session cookies (where the cookie object's `isSessionOnly` property is `true`) are local to a single process and are not shared. Session cookies are identified by the browser by the absence of an expiration date assigned to them.

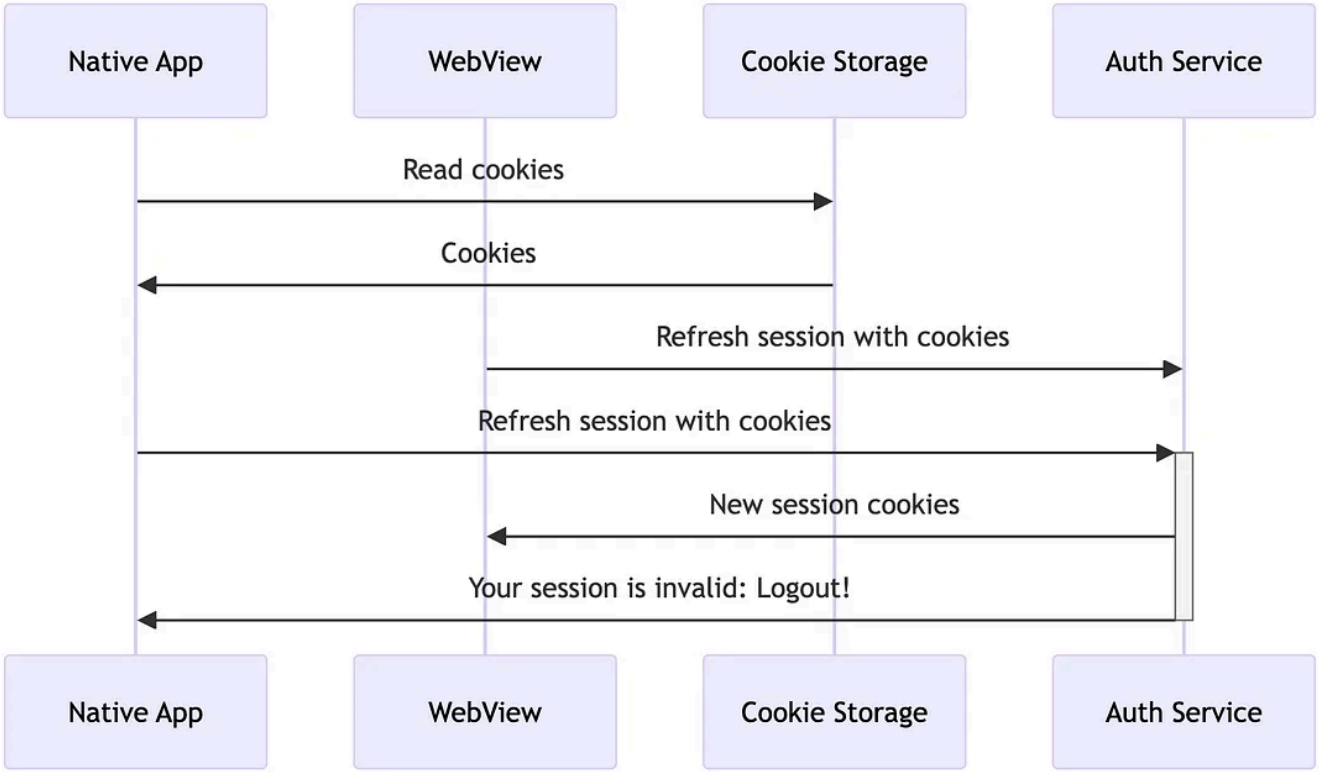
**HTTPCookieStorage | Apple Developer Documentation**

A container that manages the storage of cookies.

developer.apple.com

## Possible Race Condition

Even though we did our best, trying to synchronize the sessions, we could still face a race condition, that leads to a corrupt session:
Either while the native app starts a session refresh, the WebView kicks in with its own refresh, invalidating the session of the native app, or vice versa.




Race condition between WebView session handling and native session handling of the same session (Mermaid)

## Now the Thermonuclear Problem with WKWebViewDataStorage

Thermonuclear Explosion

Making WKWebViewDataStorage the single point of truth in the combination that we have best control over native requests, seemed to be a very good decision, BUT…

**WKWebView sometimes lose cookies a… | Apple Developer Forums**

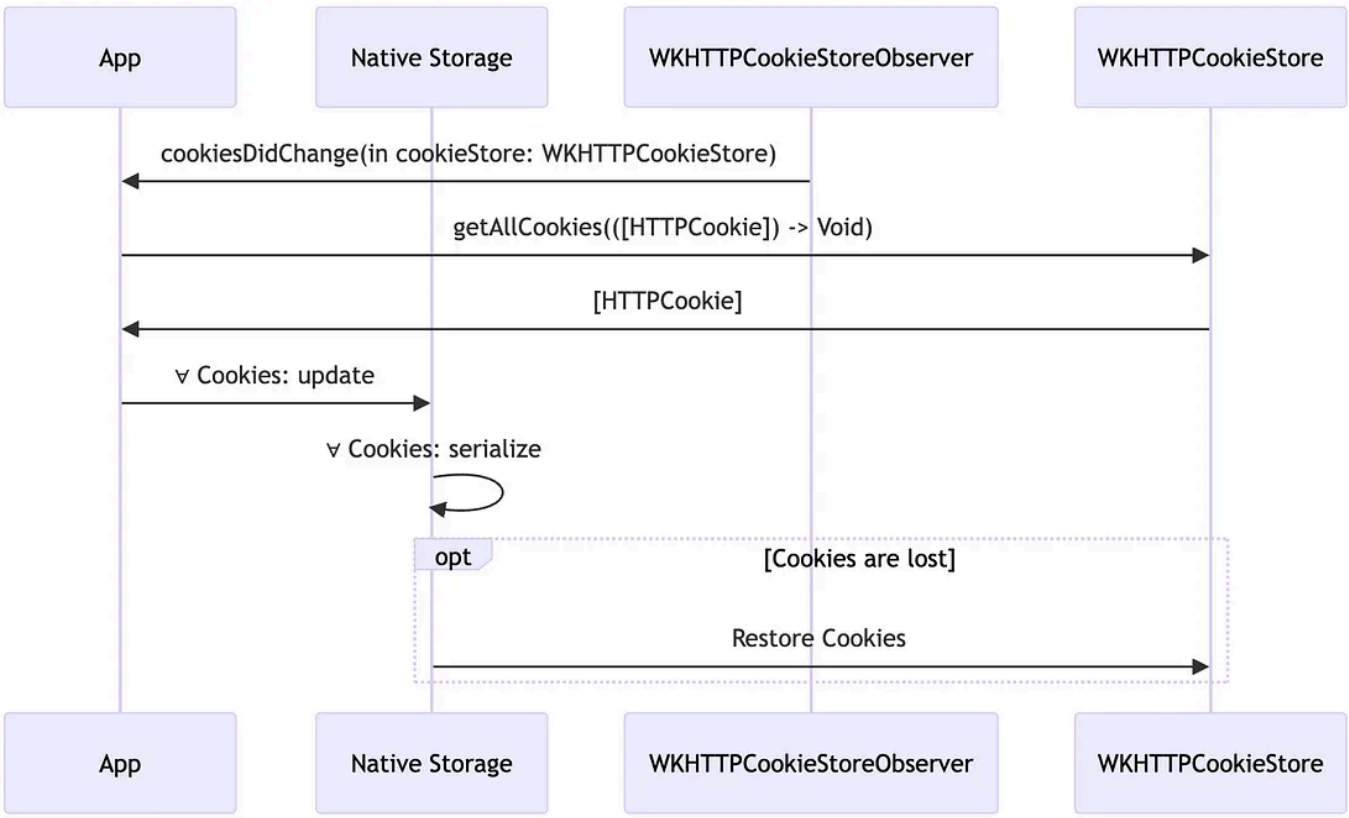
I embedded a WKWebView in my SwiftUI app, and loaded a list of HTTPCookie into a WKWebViewConfiguration object: var…

forums.developer.apple.com

> Even though we are using the default WKWebView configuration with default persistence of data, the WKWebViewDataStorage randomly loses it's cookies, resulting in a logout of users!!!

### Workaround: Native Persistence Layer for Cookies

As a workaround we therefore introduced a second native security net, always making sure that in case the cookies are lost by WKWebSiteDataStorage, we can restore the session natively and passing the cookies to the WKWebSiteDataStorage again.




Native persistence cookie layer (Mermaid)

> *Looks nice — but neither HTTPCookie nor it'S properties are not conforming to Codable…*

You could try to write a Codable cookie wrapper, that persist all known cookie properties, but then you might miss some important cookie properties, since not all possible cookie properties have a specific HTTPCookiePropertyKey assigned, e.g. HttpOnly.

**HTTPCookiePropertyKey | Apple Developer Documentation**

Constants that define the supported keys in a cookie attributes dictionary.

developer.apple.com

The properties of an HTTPCookie are defined as:

```
[HTTPCookiePropertyKey: Any]?
```

Where `HTTPCookiePropertyKey` is a `String` alias — but `Any` could be anything. We now face type safety issues:

Using `Any` removes any type information associated with the values in the dictionary. This makes it difficult to know the actual types of the stored values when retrieving them from user defaults. Without proper type safety, you might encounter runtime errors when trying to access or use the retrieved values, as they could be of unexpected types.

Luckily for us the NSKeyedArchiver supports all the types that we are facing in the cookie properties; even thought it contains some uncertainties about type safety, it did the job for us.

> *Before we checked out the JSONSerialization, which failed with type exception…*

| Data Type | NSPropertyListSerialization | JSON Serialization | NSKeyedArchiver |
|---|---|---|---|
| Basic types (String, Int, Double, Bool) | Yes | Yes | Yes |
| Arrays | Yes | Yes (limited to basic types) | Yes |
| Dictionaries | Yes (limited to specific key types) | Yes (limited to basic types as values) | Yes |
| Sets | No | No | No |
| Dates | Yes (Date, TimeInterval) | Yes (as ISO8601 string) | Yes |
| Data | Yes | No | Yes |
| Custom objects | Limited (must conform to NSCoding or implement custom logic) | No | Yes (must support NSCoding protocol) |
| Enums | Yes (must conform to RawRepresentable) | Limited (depends on underlying type) | Yes (must support NSCoding protocol) |

Data Types Supported by Serialization Methods (Gemini)

## Summary

> The root of all trouble is the fact, that practically two independent clients — the app and the WebView context inside the app — are sharing the same session.
>
> To make things even worse, additionally the WKWebView is randomly losing its cookies!

It was way harder than expected, to synchronize the sessions, and even with the solution provided, in **many** cases there can be race conditions when reading and writing of cookies happen in parallel by native or WKWebView actions – besides the catastrophe, when WKWebView loses its cookies!

iOS provides very little control over the WKWebView behaviour, but we will keep an eye on future improvements.

Back in time UIWebView shared the app level Cookie storage (HTTPCookieStorage) with other network API's like NSURLSession and NSURLConnection, which made the handling much easier.

Why WKHTTPCookieStore behaves so differently and is therefore much harder to manage, might have multiple reasons, among sandboxing the WebViews from the sensitive private data of native requests for additional security or the internal structure of the WKWebView does not allow for a default full automatic sync.

Feedback from Ivan Lisovyi, Acting iOS Staff Engineer at OLX Group

> *Well, I spent a few months fixing various authentication-related issues in the OLX iOS app, including cookies being cleaned periodically without any action from the app. What a coincidence!* 😅
>
> *TL;DR: As you pointed out at the end of the article, cookies are unreliable, and any logic relying on WKWebView doing something is fragile. Consequently, we ended up rewriting the entire authentication flow almost from scratch. I'm uncertain if there's a good solution for the problem you're encountering.*

## Future Options

### Option A: Duplicate and isolate the session

The basic problem that we are facing is that the native app and the WebView inside of the app, act practically as independently clients.

Instead of trying to keep the two session handlings (native and WebView) in sync, we can try to duplicate and isolate the session for each client, by using a session proxy:

1. If the native app has to open an URL, instead of directly assessing the URL, it passes the current session cookies and the target URL to a session proxy.
2. The session proxy will clone the session, to isolate the WebView session from the native session, and then will redirect to target page with cloned session set.
3. When the WebView is closed again, the native session will be refreshed, to get any changes that might have happen, during the usage of the

WebView.

> Are you silly? Redirecting all web requests and the requirement for this proxy is insane!

Well, you are right… but it was still an option…

### Option B: Pass session control to the native app

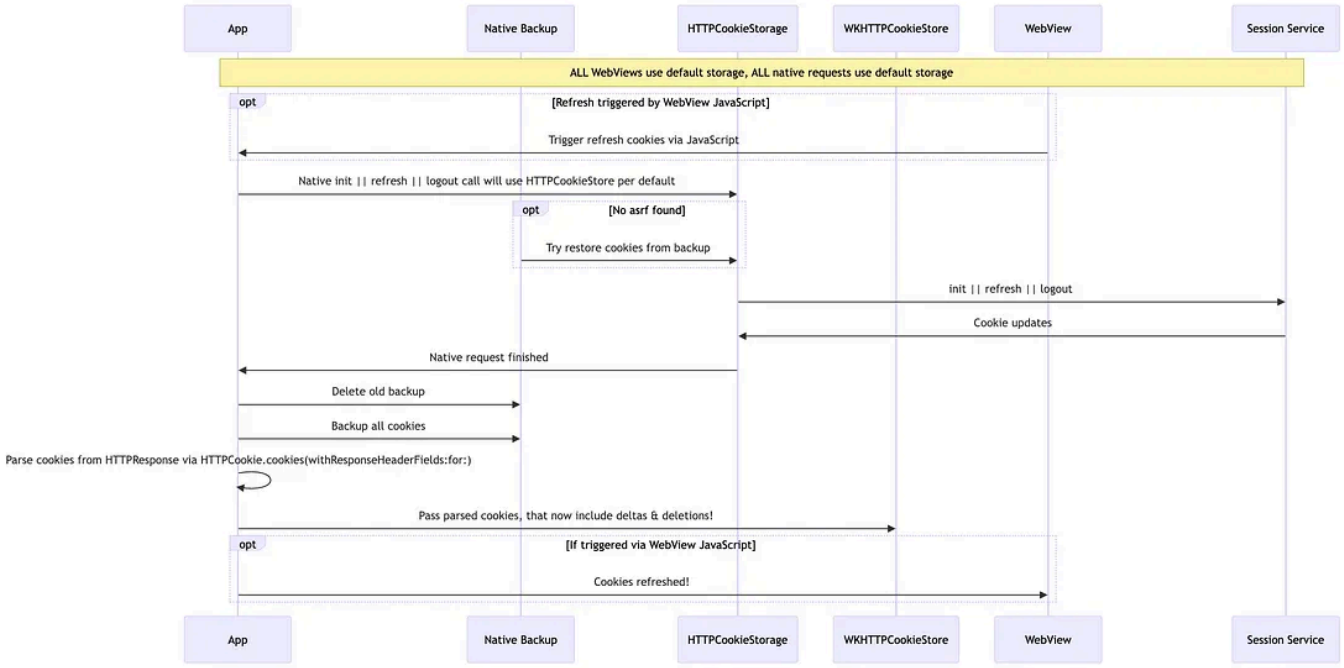
At our internal session management, inside the WebView the session can be refreshed via JavaScript.
To avoid race conditions between refresh calls inside the WebView and the native calls, plus making sure that the WKWebViewDataStorage is not losing its cookies, by managing them with native persistence, we can pass the control for the session management to the native app.

So if a web page is running inside the app inside a WebView, the refresh-script can be context aware, and pass the control over session updates to the native app via WKScriptMessageHandlerWithReply.
The native app will then be triggered via the WebView script to request a session refresh and will write the updated session cookies back to the session storage of the WebView **BEFORE** the WebView executes the navigation, which could also change the cookies via HTTP headers.

> *Paul, but wait…*
> *If you are just copying the cookies over, you will miss the cookie deletions, that you will not get, if you just copy the cookies from the store!*

**Damn! Right!**

Then after the native request, we will have to parse the set-cookie headers of the response to HTTPCookies, that then also contain the cookie deletions, that either happen via setting the expiry time of cookies in the past. Then we can pass the HTTPCookies, that now also contain the deletions to the WKCookieStore.




Native controlled session and refresh trigger by JavaScript (Mermaid)

On top of that, to make sure we are even more human error safe, we might NOT pass the refresh token to the WebViews, to render it impossible for WebViews to corrupt the native session.

**WKScriptMessageHandlerWithReply | Apple Developer Documentation**

An interface for responding to messages from JavaScript code running in a webpage.

developer.apple.com

**Build web apps in WebView | Android Developers**

Binding JavaScript

developer.android.com

Finally, since we are unsure if other SDKs integrated in the app are fiddling around with the shared storage of either HTTPCookieStorage or WKCookie store, we will isolate us from the shared storage using WKWebView and private browsing, plus the ephemeral configuration for native calls.

> Hey this might work for you, but not for us!

Ya, might be the case, but then you will have to look for a custom solution suitable for your individual context…

**Which one you choose?**

**Hope you enjoyed the show!**

We hope you find our article about how to synchronizing native and WebView sessions insightful. Please don't hesitate to contact us if you have any questions, or even better, if you've discovered a more sustainable solution!

## Input from our Android Crew

Even though iOS had hard times, for Android the same thing wasn't easy peasy as well…

- It was very hard to implement correct (that's not a problem now, but it was)
- It creates very high coupling between mobile and web
  CookieManager has very limited APIs, e.g. we can't do an offline logout now because there is no delete cookies API
- Biggest problem now: two components are responsible for token refresh, native and web, which can cause multiple refreshes at the same time leading to corrupt session
- The Chromium browser on Android has some issues as well…

> *Android cannot guarantee, when starting an app, that the previous copy of that app's process has actually exited. It tries, and it tries harder in Android 11+, but there's one case that can't be fixed: when one of the threads in the old process is stuck in the kernel in uninterruptible sleep or an infinite loop, usually due to a device driver bug. The thread won't be terminated even by SIGKILL and so the process it's part of will stick around forever, holding all its open file descriptors and thus their corresponding advisory locks. :(*

> *This is very rare in general, but it's not evenly distributed: the bugs that get threads stuck are device and OS build specific and so while it may *never* happen for an "average" user, the unlucky users whose devices have a relevant bug (and the unlucky app developers whose users use those devices) may be in a worse state. Some apps may also be doing something that *triggers* one of these bugs in some situation, which would make it especially bad for that app.*

**Chromium**

Threading problem bug report

issues.chromium.org

App Development | Oauth | Jwt | IOS | Webview

## Published in Axel Springer Tech

399 Followers · Last published 2 days ago

Follow

Tech and product folks writing about their work across the Axel Springer group: https://www.axelspringer.com/

## Written by Paul Hackenberger

96 Followers · 6 Following

Follow

Team Lead Native Mobile Development https://www.linkedin.com/in/paul-hackenberger/